C4

ActivateTrailQueue 54 (EDMRESchedApi.cc)
AddWIToTrailQueue.....66 (EDMRESchedApi.cc)
DeactivateTrailQueue 55 (EDMRESchedApi.cc)
DebugLogFds.....34 (RSLwisvr.c)
DecrementRunningWI 61 (EDMRESchedApi.cc)
DetermineGlobalDriveUse....31 (RSLwisvr.c)
ExecuteWorkItemRestore 6 (RSLstart.c)
FindTrailQueueOfWI.....67 (EDMRESchedApi.cc)
GenerateTrailQueues 51 (EDMRESchedApi.cc)
GetRunningWI.....63 (EDMRESchedApi.cc)
GetTQDriveConcurrency 57 (EDMRESchedApi.cc)
GetTQDrivesAcquired.....59 (EDMRESchedApi.cc)
HandleWorkItemRestoreResults 25 (RSLwisvr.c)
IncrementRunningWI.....60 (EDMRESchedApi.cc)
InitiateWorkItemRestore 23 (RSLwisvr.c)
InterpretWorkItemRestoreResults...33 (RSLwisvr.c)
LockScheduleMutex 42 (EDMRESchedApi.cc)
LookupSchedWI.....44 (EDMRESchedApi.cc)
LookupTrailObject 46 (EDMRESchedApi.cc)
NewSchedWI.....48 (EDMRESchedApi.cc)
NewTrailObject 47 (EDMRESchedApi.cc)
PopWIFromTrailQueue.....64 (EDMRESchedApi.cc)
PurgeTrailQueue 53 (EDMRESchedApi.cc)
RSTSL_Start.....2 (RSLstart.c)
RunCleanupRestore 9 (RSLstart.c)
RunExecutionOverrideRestore...11 (RSLstart.c)
RunPrepareRestore 7 (RSLstart.c)
RunWorkItemRestores.....15 (RSLwisvr.c)
RunWorkItemRestoresForTrail 29 (RSLwisvr.c)
Select.....22 (RSLwisvr.c)
SendRunningWorkItemsQuit 32 (RSLwisvr.c)
SetRunningWI.....62 (EDMRESchedApi.cc)
SetTQDriveConcurrency 56 (EDMRESchedApi.cc)
SetTQDrivesAcquired.....58 (EDMRESchedApi.cc)
UnlockScheduleMutex 43 (EDMRESchedApi.cc)
alwaysFalse.....8 (RSLstart.c)
findTrail 50 (EDMRESchedApi.cc)
test_fd.....35 (RSLwisvr.c)
test_fd_hup 36 (RSLwisvr.c)

Wed Jan 02 16:32:10 2008 Function Index Page 2

Wed Jan 02 16:32:10 2008 Function Index Page 1

Wed Jan 02 16:32:10 2008 Function Index Page 2

RSLstart.c **1**
ExecuteWorkItemRestore.....6
RSTSL_Start 2
RunCleanupRestore.....9
RunExecutionOverrideRestore 11
RunPrepareRestore.....7
alwaysFalse 8
RSLwisvr.c **13**
DebugLogFds 34
DetermineGlobalDriveUse....31
HandleWorkItemRestoreResults 25
InitiateWorkItemRestore....23
InterpretWorkItemRestoreResults 33
RunWorkItemRestores.....15
RunWorkItemRestoresForTrail 29
Select.....22
SendRunningWorkItemsQuit 32
test_fd.....35
test_fd_hup 36
EDMRESchedApi.cc **41**
ActivateTrailQueue 54
AddWIToTrailQueue.....66
DeactivateTrailQueue 55
DecrementRunningWI.....61
FindTrailQueueOfWI 67
GenerateTrailQueues.....51
GetRunningWI 63
GetTQDriveConcurrency.....57
GetTQDrivesAcquired 59
IncrementRunningWI.....60
LockScheduleMutex 42
LookupSchedWI.....44
LookupTrailObject 46
NewSchedWI.....48
NewTrailObject 47
PopWIFromTrailQueue.....64
PurgeTrailQueue 53
SetRunningWI.....62
SetTQDriveConcurrency 56
SetTQDrivesAcquired.....58
UnlockScheduleMutex 43
findTrail.....50

Wed Jan 02 16:32:10 2008 File Index Page 2

Wed Jan 02 16:32:10 2008 File Index Page 1

Wed Jan 02 16:32:10 2008 File Index Page 2

```
/*****************************************************************************
                                                                          ******
**
** File Name:   RSLstart.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**       The intent of the contents of this file is to implement the
**       functions the control execution of the restore for the
                                                              Restore Service
**       Library.
**
**       These functions are provided to allow:
**      - creation of submit objects,
                                    which define the set of objects to be
**        restored and the scripts to be run before and after
                                                                 restoration,
**      - starting the restoral of a submit object.
**
** The following functions comprise restoral management:
**
**      RSTSL_Start
**
**
** Compile-Time Options:
**               This section must list any compile time definitions
**               which will affect this header.
**
*****************************************************************************
                                                                       *****/



/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/wait.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
```



```
#include <RSLinterns.h>
#include <RSLauxSupp.h>
#include <restore/EDMRESubmitApi.h>

#include <restore/REprogmsg.h>
#include <restore/dispatch_daemon.h>
#include <restore/EDMREProgressApi.h>

extern int RunExecutable(const boolean_ty ResetUid,
                         const int RunUid,
                         const char *starting_cwd,
                         const char *executable_name,
                         char **executable_argv,
                         char **executable_env,
                         int *run_exit_status,
                         boolean_ty *run_cancelled,
                         boolean_ty (*QuitTest)(void));

extern int RunWorkItemRestores(int, boolean_ty (*CancelRestoreTest)());

static eerrno_ty
ExecuteWorkItemRestore(int SubmitObjectID,
                       boolean_ty (*QuitTest)(void));

static eerrno_ty
RunPrepareRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int *PrepareExit);
static eerrno_ty
RunCleanupRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int runphase_status,
                  int *CleanupExit);

/*
 * #defines, structures, typedefs local to this source file
 */

#define STR_SURE(str) (str) ? str:""
#define REMOVE_NEWLINE(str)\
{\
   int rem_nl_index;\
   for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
     {\
       if(str[rem_nl_index] == '\n')\
          str[rem_nl_index] = '\0';\
     }\
}

/*****************************************************************************
 * Start
 *
 * This function begins execution of the restoral of the objects in a
 * submit object.  Its progress and requests for operator input are
 * returned via RSTSL_GetRestoreFeedback.
 *
 * Parameters:
 *
 * SubmitObjectID (
                I) - ID of the submit object which describes the restore
 * QuitTest       (I) - function to call to check for quit signal
 *
 *****************************************************************************/
eerrno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest)(
                                                                    void))
{
```

```
127 1   int ret_pre;
128 1   int ret_exec;
129 1   int ret_post;
130 1   int ret_all_ok;
131 1   int PrepareExit = 0;
132 1   int CleanupExit = 0;
133 1   boolean_ty QuitFlag = FALSE;

135 1   char asctime[32];

137 1   memset(asctime, 0, 32);

139 1   (void)time(&rcp->rc_cmd_starttime);

141 1   (void)ctime_r(&rcp->rc_cmd_starttime, asctime, 32);

143 1   rcp->rc_cmd_last_waf_time = rcp->rc_cmd_starttime;

145 1   REMOVE_NEWLINE(asctime);

147 1   rbe_log_stats(0, "Restore Started at %s.", asctime);

149 1   rbe_log_stats(0, "Restore Started application type %s.",
150 1                 (rcp->rc_backup_app == 0)
151 1                         ? "Network"
152 1                         : ((struct pluginIDdata *)(
                                        rcp-> currentPIptr-> idData))
153 1                                                          ->
                                                          name );


155 1   rbe_log_stats(0, "Restore Started of "\
156 1                 "top level object: %s. template %s, trailset %s.",
157 1                 STR_SURE(rcp->rc_top_level_object_name),
158 1                 STR_SURE(rcp->rc_template_name),
159 1                 (rcp->rc_saveset_thread) ? "Alternate": "Primary");


162 1   rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
163 1                 STR_SURE(rcp->rc_human_uidname),
164 1                 rcp->rc_human_uid, rcp->rc_human_gids[0]);

166 1   rbe_log_stats(0, "Restore Started with client destination %s.",
167 1                 STR_SURE(rcp->rc_client_hostname));

169 1   /* if not a network restore,
                        check if plugin has its own start function */

171 1   if ( rcp-> rc_backup_app != 0
172 1     && NULL != rcp-> currentPIptr-> piFuncArray[
                                          PIFuncIndexStartRestore ] )
173 2   {
174 2     setGlobalStatus( EDMRE_STATE_EXECUTE );
                                        /* set RE's internal status */
175 2     ret_exec = rcp-> currentPIptr-> piFuncArray[
                                            PIFuncIndexStartRestore ]
176 2                                           (
                                      rcp, SubmitObjectID, QuitTest);


178 2     if( QuitTest() )      /* check for abort before return */
179 3     {
180 3       rbe_log_stats( EP_RB_RECOVER_ABORT,
181 3                      "The restore was quit by the user during
                                                          execution.");
182 3       setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                        /* set RE's internal status */
```



```
183 3       return EP_RB_RECOVER_ABORT;
184 2     }

186 2     if (E_SUCCESS != ret_exec)
187 2       setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
188 2     else
189 2       setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                          /* set RE's internal status */


191 2     return( ret_exec );
192 1   }

194 1   ret_pre = RunPrepareRestore(SubmitObjectID,
195 1                               QuitTest,
196 1                               &PrepareExit);

198 1   if(EP_RB_RECOVER_ABORT == ret_pre)
199 2   {
200 2     rbe_log_stats(EP_RB_RECOVER_ABORT,
201 2                   "The restore was quit by the user during
                                                          preparation.");
202 2     setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
203 2     return EP_RB_RECOVER_ABORT;
204 1   }
205 1   if(PrepareExit != 0)
206 2   {
207 2     rbe_log_stats(EP_RB_RECOVER_PREFAILED,
208 2                   "The restore failed during preparation. Exit %d",
209 2                   PrepareExit);

211 2     setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
212 2     return (EP_RB_RECOVER_PREFAILED);
213 1   }

215 1   setGlobalStatus( EDMRE_STATE_EXECUTE );
                                          /* set RE's internal status */

217 1   ret_exec = ExecuteWorkItemRestore(SubmitObjectID,
218 1                                     QuitTest);

220 1   QuitFlag = QuitTest();          /* check for abort before cleanup */


223 1   if (E_SUCCESS == (
              ret_all_ok = ret_exec) )      /* check if any WIs failed */
224 2   {
225 2     int local_stat;
226 2     EDMStats stats;
227 2     memset( &stats, 0, sizeof(EDMStats) );

229 2     if (0 != getRestoreStatus( 0, &stats, &local_stat ))
230 3     {
231 3       rbe_log_stats(
                local_stat, "Internal error: Failed in getRestoreStatus.");
232 3       ret_exec = ret_all_ok = EP_RB_RECOVER_EXECUTEFAILED;
233 2     }
234 2     else if (stats.edm.failed)
235 3     {     /* if any workitems failed,
                                        its a failure for cleanup purposes */
236 3         if (0 == stats.edm.successful)
237 3           ret_all_ok = EP_RB_RECOVER_ALLFAIL;
238 3         else if (stats.edm.successful > stats.edm.failed)
239 3           ret_all_ok = EP_RB_RECOVER_FEWFAIL;
```

```
240 3           else
241 3             ret_all_ok = EP_RB_RECOVER_MANYFAIL;
242 2       }
243 1     }

245 1     ret_post = RunCleanupRestore(SubmitObjectID,
246 1                                  QuitTest,
247 1                                  ret_all_ok,
248 1                                  &CleanupExit);

250 1     if(QuitFlag)
251 2     {
252 2       rbe_log_stats(EP_RB_RECOVER_ABORT,
253 2                     "The restore was quit by the user during execution.
                                                                          ");
254 2       setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
255 2       return EP_RB_RECOVER_ABORT;
256 1     }

258 1     if (E_SUCCESS != ret_exec)      /* return execute status if it failed
                                                                          */
259 2     {
260 2       setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
261 2       return ret_exec;
262 1     }

264 1     if(EP_RB_RECOVER_ABORT == ret_post)
265 2     {
266 2       rbe_log_stats(EP_RB_RECOVER_ABORT,
267 2                     "The restore was quit by the user during cleanup.");
268 2       setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
269 2       return EP_RB_RECOVER_ABORT;
270 1     }

272 1     if( (CleanupExit != 0) || (E_SUCCESS != ret_post) )
273 2     {
274 2       rbe_log_stats(EP_RB_RECOVER_POSTFAILED,
275 2                     "The restore failed during cleanup. Exit %d",
276 2                     CleanupExit);

278 2       setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
279 2       return (EP_RB_RECOVER_POSTFAILED);

281 1     }

283 1     setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                          /* set RE's internal status */

285 1     return( E_SUCCESS );

287   }       /* RSTSL_Start */
```



```
292   static eerrno_ty
293   ExecuteWorkItemRestore(int SubmitObjectID,
294                          boolean_ty (*QuitTest)(void))

297 1 {
298 1     int ret_RunWItem;
299 1     sm_push();

301 1     rcp->error_message[0] = 0;

303 1     if(0 != (ret_RunWItem = RunWorkItemRestores(
                                              SubmitObjectID, QuitTest)))
304 2     {
305 2       rbe_log_stats(0,"Internal error: Failed in RunWorkItemRestores");
306 1     }

308 1     sm_pop();

310 1     if (QuitTest() == TRUE)
311 1         return EP_RB_RECOVER_ABORT;

313 1     if (ret_RunWItem != 0)
314 1         return EP_RB_RECOVER_EXECUTEFAILED;

316 1     return E_SUCCESS;
317   }
```

```
320   #define EXECUTABLE_MAX 1024
321   static eerrno_ty
322   RunPrepareRestore(int SubmitObjectID,
323                     boolean_ty (*QuitTest)(void),
324                     int *PrepareExit)
325 1 {
326 1   char **prephaseargs = NULL;
327 1   char **prephaseenv = NULL;
328 1   int GetSOstatus = 0;
329 1   char preExecutable[EXECUTABLE_MAX];
330 1   boolean_ty restore_cancelled = FALSE;

332 1   *PrepareExit = 0;

334 1   /*
335 1    * GetSOPrePhase allocates prephaseargs & prephaseenv.
336 1    * This will need to be free'ed later.
337 1    *
338 1    */

340 1   if(0 != GetSOPrePhase(SubmitObjectID,
341 1                         preExecutable,
342 1                         EXECUTABLE_MAX,
343 1                         &prephaseargs,
344 1                         &prephaseenv,
345 1                         &GetSOstatus))
346 2   {
347 2     rbe_log_stats(0,"Internal error: Failed in GetSOPrePhase");
348 2     return (EP_RB_RECOVER_FATALERR);
349 1   }

351 1   if(0 != strcmp(preExecutable, ""))
352 2   {
353 2     setGlobalStatus( EDMRE_STATE_PREPHASE );
354 2     if(-1 == RunExecutable(FALSE,
355 2                            0,
356 2                            NULL,
357 2                            preExecutable,
358 2                            prephaseargs,
359 2                            prephaseenv,
360 2                            PrepareExit,
361 2                            &restore_cancelled,
362 2                            QuitTest))
363 3     {
364 3       rbe_log_stats(
                0,"Internal error: Failed in RunExecutable for prepare.");
365 3       return (EP_RB_RECOVER_FATALERR);
366 2     }
367 2     if(TRUE == restore_cancelled)
368 2       return (EP_RB_RECOVER_ABORT);

370 1   }

372 1   return( E_SUCCESS );
373   }
```



```
375   boolean_ty alwaysFalse() { return FALSE; }
```

```
377  static eerrno_ty
378  RunCleanupRestore(int SubmitObjectID,
379                    boolean_ty (*QuitTest)(void),
380                    int runphase_status,
381                    int *CleanupExit)
382  1 {

384  1   char **postphaseargs = NULL;
385  1   char **postphaseenv = NULL;
386  1   int GetSOstatus = 0;
387  1   char postExecutable[EXECUTABLE_MAX];
388  1   boolean_ty restore_cancelled = FALSE;
389  1   boolean_ty ignore_quit=FALSE;

391  1   *CleanupExit = 0;

393  1   /*
394  1    * GetSOPostPhase allocates postphaseargs & postphaseenv.
395  1    * This will need to be free'ed later.
396  1    *
397  1    */
398  1   PurgeTrailQueue();
399  1   if(0 != GetSOPostPhase(SubmitObjectID,
400  1                          postExecutable,
401  1                          EXECUTABLE_MAX,
402  1                          &postphaseargs,
403  1                          &postphaseenv,
404  1                          &GetSOstatus))
405  2   {
406  2     rbe_log_stats(0,"Internal error: Failed in GetSOPostPhase");
407  2     return (EP_RB_RECOVER_FATALERR);
408  1   }

410  1 #define RESTORE_BREAK        "RESTORE_BREAK="
411  1 #define RESTORE_BREAK_TRUE   "RESTORE_BREAK=T"
412  1 #define RESTORE_BREAK_ERROR "RESTORE_BREAK=E"

414  1   if(0 != strcmp(postExecutable, ""))
415  2    {
416  2      /*
417  2       * If a quit has been specified, we need to tweak the
418  2       * RESTORE_BREAK environment variable if set
419  2       */
420  2      char *abort=NULL;
421  2      if(QuitTest())
422  3        {
423  3           abort=RESTORE_BREAK_TRUE;
424  2        }
425  2      else if(0!=runphase_status)
426  3        {
427  3           abort=RESTORE_BREAK_ERROR;
428  2        }

430  2      /*
431  2       * ignore_quit is set to 1 when we have already processed a BREAK
432  2       * (CANCEL from gui), and are using the environment variable
433  2       * RESTORE_BREAK_E to signal the post restore script to clean
434  2       * up form this break.  When that happens, an ignore_quit value
435  2       * of 1 will cause actual quit signals to be ignores by the
436  2       * cleanup script, since we already know (via the environment
437  2       * variable) that we are in "cleanup mode" and no further signal
438  2       * intercpetion is necessary.
439  2       */
440  2      ignore_quit=FALSE;
441  2      if(NULL!=abort && NULL!=postphaseenv)
```



```
442  3        {
443  3           int isub=0;
444  3           char *cptr;
445  3           while(cptr=postphaseenv[isub])
446  4             {
447  4               if(strncmp(postphaseenv[isub],RESTORE_BREAK,strlen(
                                                        RESTORE_BREAK))==0)
448  5                 {
449  5                   postphaseenv[isub]=esl_strdup(abort);
450  5                   ignore_quit=TRUE;
451  5                   if(NULL==postphaseenv[isub])
452  6                     {
453  6                       rbe_log_stats(
          EP_RB_RECOVER_MALLOC_FAILURE,"Allocate failed in RSLstart.c");
454  6                       return EP_RB_RECOVER_POSTFAILED;
455  5                     }
456  4                 }
457  4               isub++;
458  3             }
459  2        }

461  2     setGlobalStatus( EDMRE_STATE_POSTPHASE );
                                                  /* set RE's internal status */
462  2     if(-1 == RunExecutable(FALSE,
463  2                            0,
464  2                            NULL,
465  2                            postExecutable,
466  2                            postphaseargs,
467  2                            postphaseenv,
468  2                            CleanupExit,
469  2                            &restore_cancelled,
470  2                            ignore_quit?alwaysFalse:QuitTest))
471  3     {
472  3       rbe_log_stats(
              0,"Internal error: Failed in RunExecutable for cleanup.");
473  3       return (EP_RB_RECOVER_FATALERR);
474  2     }
475  2     if(TRUE == restore_cancelled)
476  2       return (EP_RB_RECOVER_ABORT);

478  1   }
479  1   return( E_SUCCESS );
480  }
```

```
482  static eerrno_ty
483  RunExecutionOverrideRestore(int SubmitObjectID,
484                              boolean_ty (*QuitTest)(void))
485  1 {
487  1   return( E_SUCCESS );
488  }
```



```
489  #undef EXECUTABLE_MAX
```

```
/*****************************************************************************
                                                                          ******
**
** File Name:   RSLwisvr.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions the control execution of work item restores for
                                                                        Restore
**      Service  Library.
**
**      These functions are provided to allow:
**
** The following functions comprise restoral management:
**
**      RunWorkItemRestores()
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************************
                                                                          *****/



#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <values.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <restore/REprogmsg.h>

/*
 * Local headers
 */

#include <RSLspexits.h>
#include <RSLremfd.h>
#include <EDMRESchedApi.h>
#include <EDMREHandleMgrApi.h>
#include <RSLinterns.h>
#include <RSLrbrmain.h>
#include <restore/EDMRESubmitApi.h>
#include <EDMREDrainApi.h>

#define STR_SURE(str) (str) ? str:""


/*
```



```
 *
 * RunWorkItemRestores
 * {
 * Set the number of drives being used for the life of the restore.
 * SetQuitFlag = FALSE
 * TrailRestoresLeft = { # of trail restores }
 * TrailRestoresRunning = 0;
 *
 *   while drive available.
 *     RunTrail -- Set drive concurrency for trail restore.
 *     TrailRestoresRunning++;
 *     while OKToRunWIForTrail
 *       StartWIRestore
 *
 *   while(1)
 *     if(QuitTest)
 *        Send WICancells
 *        SetQuitFlag = TRUE
 *     Select(from_pipe, timeout {5 seconds})
 *     for each WI that completes
 *       interperate return.
 *       Drain progress.
 *       Send final Progress for work item.
 *       if(WIfailed)
 *          if(OKtoReschedule && SetQuitFlag == FALSE)
 *             Add to TrailQueue
 *       if(TrailRestoreHasMoreWorkitems && SetQuitFlag == FALSE)
 *          while OKToRunWIForTrail
 *             StartWIRestore
 *       else -- RunNewTrailRestore
 *          EndTrailRestore(prevTrailQueue)
 *          TrailRestoresRunning--;
 *          TrailRestoresLeft--;
 *          while (drive available &&
 *                 SetQuitFlag == FALSE &&
 *                 TrailRestoresLeft)
 *
 *              RunTrail -- Set drive concurrency for trail restore.
 *              TrailRestoresRunning++;
 *              while (
 *                    OKToRunWIForTrail && SetQuitFlag == FALSE)
 *                StartWIRestore
 *              end while
 *           end while
 *        end else
 *    End for each WI completes
 *    if (((SetQuitFlag == TRUE) && (TrailRestoresRunning == 0))||
 *         TrailRestoresLeft == 0)
 *         return;  exit loop.
 *  end while(1)
 *
 * }
 *
 */

/* Stubs */
static int
InterpretWorkItemRestoreResults(wi_restore_results *results);

static int
test_fd(int fd);

static void
DebugLogFds(char *error_msg,
            fd_set *fds);
```

```
129  static int
130  DetermineGlobalDriveUse();

132  static int
133  test_fd_hup(int fd);

135  static int
136  FindTrailIDForWItem(int handle,
137                      int *TrailID,
138                      int *status);

140  static int
141  SendRunningWorkItemsQuit();

143  /* End Stubs */

145  static int
146  Select(int nfds,
147         fd_set *readfds,
148         fd_set *writefds,
149         fd_set *exceptfds,
150         struct timeval *timeout);


153  static int
154  HandleWorkItemRestoreResults(int FromFD,
155                               int *TrailID,
156                               wi_restore_results *results);
157  static int
158  RunWorkItemRestoresForTrail(const int TrailID,
159                              const int CountDrivesAvailable,
160                              boolean_ty (*CancelRestoreTest)(),
161                              boolean_ty *QuitFlag,
162                              int *CountDrivesInUse);

164  /*
165   * RunWorkItemRestores()
166   *
167   *  Runs a set of work item restores.
168   *
169   * Args:
170   *  SubmitObject
171   *  CancelRestoreTest().
172   *
173   * Returns: int 0 for success.
174   */
175  int
176  RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)(
                                                                        ))
177  1 {
178  1   boolean_ty QuitFlag = FALSE;  /* Has the user requested a quit.*/
179  1   boolean_ty SentQuit = FALSE;  /* Have we initiated the quit. */

181  1   int TrailRestoresRunning = 0;
                                 /* The number of trail restores running. */
182  1   int TrailRestoresLeft;
                                  /* The number of trail restores left. */
183  1   int TrailRestoresTotal;
                                 /* The number of trail restores total. */

185  1   int CountDrivesAvailable;     /* The count of drives available. */
186  1   int CountDrivesInUse = 0;     /* The count of drives in use. */

188  1   int temp_status;
189  1   int HighestActiveTrail = 0;
                              /* The trail queues are ordered from 1 to n. */
```



```
190  1                                /* This is the highest trail running
                                                                        */

192  1   if(debugmode)
193  2   {
194  2     (void)rbe_user_error(0,
195  2                          "DEBUG: Running RunWorkItemRestores.");
196  1   }
197  1   /* GenerateTrailQueues()
198  1    * Buckets the work items into trail queues.
                                                  The trail queues are sorted
199  1    * in the order which the restores should run.
200  1    */

202  1   if(0 != GenerateTrailQueues(SubmitObject,
203  1                               &TrailRestoresTotal,
204  1                               &temp_status))
205  2   {
206  2     (void)rbe_user_error(0,
207  2                          "Internal error: Cannot generate trail
                                                  queues, cannot continue.");
208  2     return -1;
209  1   }

211  1   TrailRestoresLeft = TrailRestoresTotal;

213  1   CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

215  1   if(debugmode)
216  2   {
217  2     (void)rbe_user_error(0,
218  2                          "DEBUG: RunWorkItemRestores for %d trails.",
219  2                          TrailRestoresTotal);
220  1   }
221  1   /*
222  1    * This is the start up loop to get the initial work item
223  1    * restores started.
224  1    */

226  1   QuitFlag = CancelRestoreTest();

228  1   while((CountDrivesInUse < CountDrivesAvailable) &&
229  1         (HighestActiveTrail < TrailRestoresTotal) &&
230  1         (FALSE == QuitFlag))
231  2   {
232  2     int submitObjID = 0;
233  2     int submitelementID = 0;

235  2     HighestActiveTrail++;

237  2     /*
238  2      * Activate the Trail Queue.
                                  This allows the traill queues to be used to
239  2      * determine the work item restores to run.
240  2      */
241  2     if(0 != ActivateTrailQueue(HighestActiveTrail,
242  2                                1,
243  2                                &temp_status))
244  3     {
245  3       (void)rbe_user_error(0,
246  3                            "Internal error: Cannot activate trail
                                 queues(1) for trailid %d, cannot continue.",
247  3                              HighestActiveTrail);
248  3       return -1;
249  2     }
```

```
129  static int
130  DetermineGlobalDriveUse();

132  static int
133  test_fd_hup(int fd);

135  static int
136  FindTrailIDForWItem(int handle,
137                      int *TrailID,
138                      int *status);

140  static int
141  SendRunningWorkItemsQuit();

143  /* End Stubs */

145  static int
146  Select(int nfds,
147         fd_set *readfds,
148         fd_set *writefds,
149         fd_set *exceptfds,
150         struct timeval *timeout);


153  static int
154  HandleWorkItemRestoreResults(int FromFD,
155                               int *TrailID,
156                               wi_restore_results *results);
157  static int
158  RunWorkItemRestoresForTrail(const int TrailID,
159                              const int CountDrivesAvailable,
160                              boolean_ty (*CancelRestoreTest)(),
161                              boolean_ty *QuitFlag,
162                              int *CountDrivesInUse);

164  /*
165   * RunWorkItemRestores()
166   *
167   *  Runs a set of work item restores.
168   *
169   * Args:
170   *  SubmitObject
171   *  CancelRestoreTest().
172   *
173   * Returns: int 0 for success.
174   */
175  int
176  RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)(
                                                                        ))
177  1 {
178  1   boolean_ty QuitFlag = FALSE;  /* Has the user requested a quit.*/
179  1   boolean_ty SentQuit = FALSE;  /* Have we initiated the quit. */

181  1   int TrailRestoresRunning = 0;
                                /* The number of trail restores running. */
182  1   int TrailRestoresLeft;
                                 /* The number of trail restores left. */
183  1   int TrailRestoresTotal;
                                /* The number of trail restores total. */

185  1   int CountDrivesAvailable;     /* The count of drives available. */
186  1   int CountDrivesInUse = 0;     /* The count of drives in use. */

188  1   int temp_status;
189  1   int HighestActiveTrail = 0;
                             /* The trail queues are ordered from 1 to n. */
```



```
190  1                                  /* This is the highest trail running
                                                                        */

192  1   if(debugmode)
193  2   {
194  2     (void)rbe_user_error(0,
195  2                          "DEBUG: Running RunWorkItemRestores.");
196  1   }
197  1   /* GenerateTrailQueues()
198  1    * Buckets the work items into trail queues.
                                                  The trail queues are sorted
199  1    * in the order which the restores should run.
200  1    */

202  1   if(0 != GenerateTrailQueues(SubmitObject,
203  1                               &TrailRestoresTotal,
204  1                               &temp_status))
205  2   {
206  2     (void)rbe_user_error(0,
207  2                          "Internal error: Cannot generate trail
                                                  queues, cannot continue.");
208  2     return -1;
209  1   }

211  1   TrailRestoresLeft = TrailRestoresTotal;

213  1   CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

215  1   if(debugmode)
216  2   {
217  2     (void)rbe_user_error(0,
218  2                          "DEBUG: RunWorkItemRestores for %d trails.",
219  2                          TrailRestoresTotal);
220  1   }
221  1   /*
222  1    * This is the start up loop to get the initial work item
223  1    * restores started.
224  1    */

226  1   QuitFlag = CancelRestoreTest();

228  1   while((CountDrivesInUse < CountDrivesAvailable) &&
229  1         (HighestActiveTrail < TrailRestoresTotal) &&
230  1         (FALSE == QuitFlag))
231  2   {
232  2     int submitObjID = 0;
233  2     int submitelementID = 0;

235  2     HighestActiveTrail++;

237  2     /*
238  2      * Activate the Trail Queue.
                                  This allows the traill queues to be used to
239  2      * determine the work item restores to run.
240  2      */
241  2     if(0 != ActivateTrailQueue(HighestActiveTrail,
242  2                                1,
243  2                                &temp_status))
244  3     {
245  3       (void)rbe_user_error(0,
246  3                            "Internal error: Cannot activate trail
                                queues(1) for trailid %d, cannot continue.",
247  3                            HighestActiveTrail);
248  3       return -1;
249  2     }
```

```
251 2     /*
252 2      * This sets the number of drives and media access concurrency for
                                                              trail restores.
253 2      * i.e. The count of running work item restores for this trail.
                                                            Today this is one.
254 2      */
255 2     if(0 != SetTQDrivesAcquired(HighestActiveTrail, 1, &temp_status))
256 3     {
257 3       (void)rbe_user_error(0,
258 3                            "Internal error: Cannot set drive acquired(
                1) for trailid %d, cannot continue.", HighestActiveTrail);
259 3       return -1;
260 2     }

262 2     if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                          HighestActiveTrail,
263 2
                                                        CountDrivesAvailable,
264 2
                                                          CancelRestoreTest,
265 2                                                         &QuitFlag,
266 2
                                                        &CountDrivesInUse)))
267 3     {

269 3       /* RunWorkItemRestoresForTrail does its own error logging. */
270 3       return -1;
271 2     }

273 2     if(temp_status == 0)
274 3     {
275 3       (void)rbe_log_stats(0,
276 3                           "Trail %d restore had no work item to run(
                                                    1).", HighestActiveTrail);
277 3       /* more work may be need to recover from this error condition.
                                                                        */
278 2     }
279 2     if(temp_status > 0)
280 3     {
281 3       TrailRestoresRunning++;
282 2     }
283 1   } /* End while() initial startup loop */

285 1   while(1)
286 2   {
287 2     int HighestFd = 0;
288 2     fd_set WorkItemFromFds;
289 2     int retStatus;
290 2     struct timeval timeout = {5, 0};

292 2     if((QuitFlag) && (!SentQuit))
293 3     {
294 3       (void)rbe_log_stats(0,
295 3                           "Restore was quit by user. Quitting restore,
                                              this could take a while.");

297 3       SendRunningWorkItemsQuit();
298 3       SentQuit = TRUE;
299 2     }

301 2     if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
302 3     {
303 3       (void)rbe_user_error(0,
304 3                            "Internal error: Cannot get auxproc result
                                                  fds, cannot continue.");
```



```
306 3       return -1;
307 2     }
308 2   #if 0
309 2     if(debugmode)
310 3     {
311 3       DebugLogFds("The file descriptors to wait on are ",
312 3                   &WorkItemFromFds);
313 2     }
314 2   #endif
315 2     if(0 > (retStatus = Select(HighestFd + 1,
316 2                                &WorkItemFromFds,
317 2                                NULL, NULL,
318 2                                &timeout)))
319 3     { /* error */
320 3       (void)rbe_user_error(RBRECOVER_MKERR(errno),
321 3                            "Internal error: Cannot get auxproc result
                                                  fds, cannot continue.");

323 3       return -1;
324 2     }

326 2     else if (0 == retStatus)
327 3     { /* timed out */

329 3       QuitFlag = CancelRestoreTest();

331 2     }

333 2     else
334 3     { /* Available fds */

336 3       int ReadyFds = retStatus;
337 3       int FoundFds = 0;
338 3       int index;

340 3       if(debugmode)
341 4       {
342 4          DebugLogFds("The file descriptors ready to read are ",
343 4                      &WorkItemFromFds);
344 3       }

346 3       /*
347 3        * If there are available fds then we may want to
348 3        * schedule the next work item restore. We should
349 3        * check if the user initiated a quit.
350 3        */
351 3       QuitFlag = CancelRestoreTest();

353 3       for(index = 0;
354 3           (index < (HighestFd + 1)) && (FoundFds < ReadyFds);
355 3           index++)
356 4       {
357 4          int StartWorkItemforTrail = 0;
358 4          if(FD_ISSET(index, &WorkItemFromFds))
359 5          {
360 5            int TrailID;
361 5            int TrailAcquired;
362 5            wi_restore_results results;
363 5            FoundFds++;

365 5            memset(&results, 0, sizeof(wi_restore_results));

367 5            if(0 != HandleWorkItemRestoreResults(index,
368 5                                                 &TrailID,
369 5                                                 &results))
370 6            {
```

```
371 6             /* HandleWorkItemRestoreResults will do its own logging!
                                                                        */
372 6             return -1;
373 5           }
374 5           /*
375 5            * This is where we may want to retry the Work Item
376 5            * Based on if it passes or fails
377 5            */

379 5           CountDrivesInUse--;



382 5           if (0 > (StartWorkItemforTrail =
383 5                    RunWorkItemRestoresForTrail(TrailID,
384 5                                                CountDrivesAvailable,
385 5                                                CancelRestoreTest,
386 5                                                &QuitFlag,
387 5                                                &CountDrivesInUse)))
388 6           {
389 6             /* RunWorkItemRestoresForTrail does its own logging. */
390 6             return -1;
391 5           }

393 5           else if (StartWorkItemforTrail == 0)
394 5                /* 0 work items started aboved,
395 5                 * lets check to see if this is the last work item
                                                                  left for
396 5                * this trail
397 5                 */
398 6           {
399 6             int wiCount;

401 6             if(0 != GetRunningWI(TrailID, &wiCount, &temp_status))
402 7             {
403 7               (void)rbe_user_error(0,
404 7                                    "Internal error: Cannot determine
          number of running work items for trail, cannot continue.");

406 7               return -1;
407 6             }
408 6             if(debugmode)
409 7             {
410 7               (void)rbe_user_error(0,
411 7                                    "DEBUG: RunWorkItemRestores no
                                   more work items left for trailid %d ,
                                 but %d wiCount workitem still running.",
412 7                                    TrailID, wiCount);
413 6             }
414 6             /* Testing for No work items left running or started
415 6              * For this trail.
416 6              */

418 6             if((0 == wiCount) && (0 == StartWorkItemforTrail))
419 7             {

421 7               TrailRestoresRunning--;
422 7               TrailRestoresLeft--;

424 7               if(0 != DeactivateTrailQueue(TrailID, &temp_status))
425 8               {
426 8                 (void)rbe_user_error(0,
427 8                                      "Internal error: Cannot
    deactivate trail queue for trailid %d, cannot continue.", TrailID);

429 8                 return -1;
430 7               }
```



```
432 7             /* This test is to determine,
                                          if a trail restore finished,
433 7              * there may be another not yet started trail,
                                                          If we have
434 7              * drive available the next trail restore will be
                                                              started.
435 7              */
436 7             if ((0 != TrailRestoresLeft) &&
437 7                 (HighestActiveTrail < TrailRestoresTotal) &&
438 7                 (CountDrivesInUse < CountDrivesAvailable))
439 8             {

441 8               HighestActiveTrail++;

443 8               if(0 != ActivateTrailQueue(HighestActiveTrail,
444 8                                          1,
445 8                                          &temp_status))
446 9               {
447 9                 (void)rbe_user_error(0,
448 9                                      "Internal error: Cannot
      activate trail queue(2) for trailid %d, cannot continue.",
449 9                                      HighestActiveTrail);
450 9                 return -1;
451 8               }

453 8               if(0 != SetTQDrivesAcquired(
                              HighestActiveTrail, 1, &temp_status))
454 9               {
455 9                 (void)rbe_user_error(0,
456 9                                      "Internal error: Cannot set
      drive acquired(2) for trailid %d, cannot continue.",
457 9                                      HighestActiveTrail);
458 9                 return -1;
459 8               }

461 8               if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                    HighestActiveTrail,
462 8
                                                  CountDrivesAvailable,
463 8
                                                    CancelRestoreTest,
464 8
                                                              &QuitFlag,
465 8
                                                   &CountDrivesInUse)))
466 9               {
467 9                 /* RunWorkItemRestoresForTrail does its own logging.
                                                                        */
468 9                 return -1;
469 8               }
470 8               if(temp_status == 0)
471 9               {
472 9                 /* If this Trail Had no work items we
473 9                  * Should attempt to run the next trails
474 9                  * work items here. This would be an internal
475 9                  * error if a trail queue had no work item
476 9                  * restores.
477 9                  */
478 9                 (void)rbe_log_stats(0,
479 9                                     "Internal error: Trail %d
    restore had no work item to run(1).", HighestActiveTrail);
480 9                 return -1;
481 8               }
482 8               if(temp_status > 0)
483 9               {
```

```
                    /* IF at least on work item was started for this
                                                                trail,
                     * then we have started a new trail.
                     */
                    TrailRestoresRunning++;
                  }
                }

              }
            }
          }
        } /* end for() */

      } /* else Available fds */

      /*
       * Terminate the loop if either
       * ----------------------------
       * 1) Sent the work items the quit AND
       *    No Trail restores a running.
       * OR
       * 2) No more Trail restores are left.
       */

      if(((0 == TrailRestoresRunning) && (SentQuit)) ||
         (0 == TrailRestoresLeft))
      {
        break;
      }

    }  /* end while(1) */
    if((0 == TrailRestoresRunning) && (SentQuit))
    {
      (void)rbe_log_stats(0,
                          "Restore was quit by user.
                                                Work item restore quit.");
    }
    return 0;
}
```



```

/* Functions needed
SendRunningWorkItemsQuit();
InterpretWorkItemRestoreResults();
*/

static int
Select(int nfds,
       fd_set *readfds,
       fd_set *writefds,
       fd_set *exceptfds,
       struct timeval *timeout)
{
  int retSelect;

  do
  {
    retSelect = select(nfds,
                       readfds,
                       writefds,
                       exceptfds,
                       timeout);

  } while ((-1 == retSelect) && (EINTR == errno));

  return retSelect;
}
```

```
550     eperrno
551     InitiateWorkItemRestore(const int SubmitObjID,
552                             const int SubmitElemID)
553   1 {
554   1   struct auxproc AuxprocVitals;
555   1   eerrno_ty StartupAPResults = EXIT_FAILURE;
556   1   time_t StartTime;
557   1   int TempStatus;
558   1   char junk_executable[1024];
559   1   char **junk_argv;
560   1   char **AP_env = NULL;
561   1   int SOstatus;
562   1   char clientName[256] = "";
563   1   int clientPort;
564   1   int status;
565   1   time_t EndTime;
566   1   /* Lets see if there are any environment variables to set.
567   1    *  The restore of the output variables are ingnored.
568   1    */
569   1   if(E_SUCCESS != GetSOExecutionPhase(SubmitObjID,
570   1                                       junk_executable, 1024,
571   1                                       &junk_argv,
572   1                                       &AP_env,
573   1                                       &SOstatus))
574   2   {
575   2     (void)rbe_user_error(0,
576   2                          "Internal Error: Could not get environment
                                                                  variables.");
577   2     return -1;
578   1   }

580   1   if (E_SUCCESS == GetSERcmdConnect(SubmitObjID,
581   1                                     SubmitElemID,
582   1                                     clientName, 256,
583   1                                     & clientPort,
584   1                                     &SOstatus))
585   2   {
586   2    StartupAPResults = StartupAuxprocess(0 /* XXX */,
587   2                                         &AuxprocVitals,
588   2                                         AP_env,
589   2                                         clientName,
590   2                                         clientPort);
591   1   }
592   1   else
593   2   {
594   2     (void)rbe_user_error(0,
595   2                          "Internal Error: Could not get Remote Client name
                                                                          &"
596   2                          "port to connect.");
597   2     return -1;
598   1   }

600   1   if(E_SUCCESS != StartupAPResults)
601   2   {
602   2     /* StartupAuxprocess does its own logging. */
603   2     return -1;
604   1   }

606   1   /*
607   1    * We need to close the bulk fd. This file descriptor
608   1    * is not being used any more. If we do not close it
609   1    * here we will have a file descriptor leak because
610   1    * we won't be able to determine what it was when the
611   1    * work item completes.
```



```
612   1    */

614   1   close(AuxprocVitals.xp_fd_bulk_to_x);

616   1   time(&StartTime);

618   1   if(0 != newHandleSet(AuxprocVitals.xp_fd_to_x,
619   1                        AuxprocVitals.xp_fd_from_x,
620   1                        AuxprocVitals.xp_fd_prog_from_x,
621   1                        SubmitObjID,
622   1                        SubmitElemID,
623   1                        AuxprocVitals.xp_pid,
624   1                        StartTime,
625   1                        &TempStatus))
626   2   {
627   2     (void)rbe_user_error(
                    0, "Internal Error: Could not register handle set.");
628   2     return -1;
629   1   }


632   1   if(0 > StartWorkItemRestore(rcp,
633   1                               &AuxprocVitals,
634   1                               SubmitObjID,
635   1                               SubmitElemID))
636   2   {
637   2     /*
638   2      * StartWorkItemRestore does logging if initialization fails
639   2      */
640   2      (void)rbe_user_error(
                  0, "Error in StartWorkItemRestore  SubmitObjID %d,"
641   2                       "SubmitElemID %d ",SubmitObjID,
                                                        SubmitElemID);
642   2      /*
643   2       * the following code kills auxproc when recx or xcpio do not
                                                                    start
644   2       * we do not want an auproc sitting around.
645   2       * if errors occur in deleteHandleSet or KillWorkItemRestore the
                                                          messages are
646   2       * logged in those calls, plus,
                                we already know there was an error and that
647   2       * is why we are doing this right now.
648   2       */
649   2      time(&EndTime);

651   2      deleteHandleSet(
 AuxprocVitals.xp_fd_from_x, EndTime, EP_RB_RECOVER_ALLFAIL, &status);

653   2      KillWorkItemRestore(
                     AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_x);

655   2      return -1;
656   1    }

658   1   return 0;

660     } /* InitiateWorkItemRestore() */
```

```
663  /*
664   *       interperate return.
665   *       Drain progress.
666   *       Send final Progress for work item.
667   *       Delete the handle set.
668   */

670  static int
671  HandleWorkItemRestoreResults(int FromFD,
672                               int *TrailID,
673                               wi_restore_results *results)
674 1{
675 1   int ret = 0;
676 1   int retries = 0;
677 1   int GetAuxprocResultsStatus;
678 1   int TempStatus;
679 1   int DrainResult;
680 1   int DrainedFD;
681 1   int wiCount;
682 1   int AuxProcPid;
683 1   int ToFD, getFromFD, ProgressFD;
684 1   time_t EndTime;
685 1   unsigned long jobstat;
686 1   int timeout = 3; /* Lets try 3 seconds */
687 1   boolean_ty fromFDHangUp = FALSE;

689 1   ToFD = getFromFD = ProgressFD = -1;

691 1   while(! (fromFDHangUp))
692 2   {
693 2     GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);

695 2     if(-1 == GetAuxprocResultsStatus)
696 3     {
697 3       /* GetAuxprocResults() does its own logging */

699 3      (void)rbe_user_error(0," Error in GetAuxprocResults");
700 3       return -1;
701 2     }
702 2     if(0 == GetAuxprocResultsStatus)
703 3     {
704 3         if(test_fd_hup(FromFD) == 1)
705 4          {
706 4            fromFDHangUp = TRUE;
707 3          }
708 2     }

710 2     /* The remote result are not always going to
711 2      * be set. For example if the remote command
712 2      * is not started correctly.
713 2      */

715 2     if(results->local_exit_set == TRUE)
716 3     {
717 3       break;
718 2     }
719 2     else
720 3     {
721 3       sleep(1);
722 3       test_fd(FromFD);
723 3       continue;
724 2     }
725 1   }
```



```
727 1   time(&EndTime);

729 1   if(0 != PushDrainRequest(FromFD, &TempStatus))
730 2   {
731 2     (void)rbe_user_error(0,
732 2                          "Internal error: Could not push drain
                                                request, cannot continue.");

734 2     return -1;
735 1   }
736 1   /* Lets give the progress thread a chance to drain keeping busy in
737 1    * the meanwhile.
738 1    */

740 1   if(0 != FindTrailQueueOfWI(FromFD, TrailID, &TempStatus))
741 2   {
742 2     (void)rbe_user_error(0,
743 2                          "Internal error: Could not find trail id for
                                  finished work item, cannot continue.");

745 2     return -1;
746 1   }

748 1   if(0 != DecrementRunningWI(*TrailID, &wiCount, &TempStatus))
749 2   {
750 2     (void)rbe_user_error(0,
751 2                          "Internal error: Could not decrement running
                                  work items for trail, cannot continue.");

753 2     return -1;
754 1   }

756 1   if(0 != getPID(FromFD, &AuxProcPid, &TempStatus))
757 2   {
758 2     (void)rbe_user_error(0,
759 2                          "Internal error: Could not get auxproc pid
                                          for work item, cannot continue.");

761 2     return -1;
762 1   }

764 1   if(0 != getHandleSet(
                    FromFD, &ToFD, &getFromFD, &ProgressFD, &TempStatus))
765 2   {
766 2     (void)rbe_user_error(0,
767 2                          "Internal error: Could not get auxproc file
                                descriptors for work item, cannot continue.");

769 2     return -1;
770 1   }
771 1   if(FromFD != getFromFD)
772 2   {
773 2     (void)rbe_user_error(0,
774 2                          "Internal error:  mismatch on from file
                                descriptors for work item, cannot continue.");
775 2     return -1;
776 1   }

778 1   while (0 != (ret = PopDrainResult(timeout,
779 1                                     &DrainResult,
780 1                                     &DrainedFD,
781 1                                     &TempStatus)) && retries < 3)
782 1      retries++;

784 1   if (ret != 0)
785 2   {
```

```
786 2     (void)rbe_user_error(0,
787 2                          "Internal error: Could not pop drain results,
                                                        cannot continue.");


789 2     return -1;
790 1   }

792 1   /*
793 1    *       Send final Progress for work item. XXX
794 1    */

796 1   /* Translate the local and remote error statuses
797 1    * to an eperrno value:
798 1    */

800 1   if(0 != results->local_exit_status)     /* use local error, if any */
801 1     switch (results->local_exit_status)
802 2     {
803 2     case XG_EXIT_ALLFAIL:
804 2       jobstat = EP_RB_RECOVER_ALLFAIL;
805 2       break;
806 2     case XG_EXIT_MANYFAIL:
807 2       jobstat = EP_RB_RECOVER_MANYFAIL;
808 2       break;
809 2     case XG_EXIT_FEWFAIL:
810 2       jobstat = EP_RB_RECOVER_FEWFAIL;
811 2       break;
812 2     case SPEXIT_REMOTE_STDERR_PROTOCOL:
813 2     case SPEXIT_REMOTE_STDERR_FAIL:
814 2       jobstat = EP_RB_RECOVER_CLIENT_STDERR_FAIL;
815 2       break;
816 2     case XG_EXIT_STOPPED:         /* treat like signal */
817 2     default:      /* check for signal termination vs all generic
                                                          failures */
818 2       if ( XG_EXIT_SIGBASE < results->local_exit_status
819 2         || XG_EXIT_STOPPED == results->local_exit_status )
820 3       { /* killed by signal or stopped: separate error for sigpipe */
821 3         if (XG_EXIT_SIGBASE + SIGPIPE == results->local_exit_status)
822 3           jobstat = EP_RB_RECOVER_SERVER_SIGPIPE;
823 3          else
824 3            jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
825 2       }
826 2       else
827 3       { /* generic server failure, unless client failed too */
828 3         jobstat = EP_RB_RECOVER_SERVERFAIL;
829 3         if (0 != results->remote_exit_status)
830 3           jobstat = EP_RB_RECOVER_BOTHFAIL;
831 2       }
832 1     }
833 1   else if(0 != results->remote_exit_status)
834 1     jobstat = EP_RB_RECOVER_CLIENTFAIL;
835 1   else
836 1     jobstat = E_SUCCESS;

838 1   if( (0 != results->remote_exit_status) ||
839 1       (0 != results->local_exit_status ) )
840 2   {
841 2    int  status=0;
842 2    int  rc=0;
843 2    char *templateName=NULL;
844 2    char *winame=NULL;
845 2    char *trailsetName=NULL;

847 2    rc = getHandleSetInformation(FromFD,
848 2                                 &templateName,
849 2                                 &winame,
```



```
850 2                                &trailsetName,
851 2                                &status);
852 2    rbe_log_stats(0, "Restore Failure of "\
853 2                  "top level object: %s, template %s.",
854 2                  STR_SURE(winame),
855 2                  STR_SURE(templateName));
856 2    free(templateName);
857 2    free(winame);
858 2    free(trailsetName);
859 1   }

861 1   if( 0 != deleteHandleSet(FromFD, EndTime, jobstat, &TempStatus))
862 2   {
863 2         (void)rbe_user_error(0,
864 2                              "Internal error: Could not delete Handle
                                                        Set, cannot continue.");


866 2     return -1;
867 1   }

869 1   if(0 != KillWorkItemRestore(AuxProcPid,
870 1                               -1 /* Hack this arg is not needed yet
                                                              cmd_to */))

871 2   {
872 2     (void)rbe_user_error(0,
873 2                          "Internal error: Could not kill finished
                                                    auxproc, cannot continue.");

874 2     return -1;
875 1   }

877 1   close(ToFD);
878 1   close(FromFD);
879 1   close(ProgressFD);

881 1   if(debugmode)
882 2   {
883 2     (void)rbe_user_error(0,
884 2                          "DEBUG: HandleWorkItemRestoreResults Auxproc(
          PID %d) just finished for trailid %d work items left = %d.",
885 2                          AuxProcPid,
886 2                          *TrailID,
887 2                          wiCount);

889 2     (void)rbe_user_error(0,
890 2                          "DEBUG: HandleWorkItemRestoreResults Auxproc(
           PID %d) results are local: %d: setP:%s remote: %d set:%s.",
891 2                          AuxProcPid,
892 2                          results -> local_exit_status,
893 2                          (
                            results -> local_exit_set) ? "TRUE": "FALSE" ,
894 2                          results -> remote_exit_status,
895 2                          (
                           results -> remote_exit_set) ? "TRUE": "FALSE");



898 1   }
899 1   return 0;
900   } /* End HandleWorkItemRestoreResults() */
```

```
 904  /*
 905   * RunWorkItemRestoresForTrail()
 906   *
 907   * Description
 908   *  This function starts all the work item for the
 909   *  trail. For no this is set to one but concurrency
 910   *  will can be supportted.
 911   *
 912   * Args:
 913   *  (I) TrailID -- The id for this trail.
 914   *  (I) CountDrivesAvailable -- the total drives available to restore.
 915   *  (O) QuitFlag -- indicatate whether the user has quit the restore.
 916   *  (O) CountDrivesInUse -- The count of trails in use by restore.
 917   *
 918   * Return int
 919   *  if -1 then an error has occurred.
 920   *  if 0 or greater then the number of trail restores started will be
 921   *  returned.
 922   *
 923   */
 924  static int
 925  RunWorkItemRestoresForTrail(const int TrailID,
 926                              const int CountDrivesAvailable,
 927                              boolean_ty (*CancelRestoreTest)(),
 928                              boolean_ty *QuitFlag,
 929                              int *CountDrivesInUse)
 930 1{
 931 1   int DriveAcquiredForTrail;
 932 1   int DriveConcurencyForTrail;
 933 1   int submitObjID;
 934 1   int submitelementID;
 935 1   int popResults = 0;
 936 1   int temp_status;
 937 1   int CountOfWorkItemRestoreStarted = 0;
 938 1   int wiCount;

 940 1   while(1)
 941 2   {



 944 2     (*CountDrivesInUse)++;

 946 2     if((0 != (popResults = PopWIFromTrailQueue(TrailID,
 947 2                                                &submitObjID,
 948 2                                                &submitelementID,
 949 2                                                &temp_status))) &&
 950 2        (SCHED_NO_MORE_JOBS != temp_status))
 951 3     {
 952 3       (void)rbe_user_error(0,
 953 3                            "Internal error: Cannot pop work item off
                                              trail queue, cannot continue.");
 954 3       return -1;
 955 2     }

 957 2     if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
 958 3     {
 959 3       return CountOfWorkItemRestoreStarted;
 960 2     }

 962 2     temp_status = InitiateWorkItemRestore(
                                          submitObjID, submitelementID);

 964 2     if(temp_status != 0)
```



```
 965 3     {
 966 3       /* InitiateWorkItemRestore() does its own logging */
 967 3       (void)rbe_user_error(0, "Error in InitiateWorkItemRestore,"
 968 3            "submitObjID  %d, submitelementID %d", submitObjID,
                                                          submitelementID);
 969 3       return -1;
 970 2     }

 972 2     if(0 != IncrementRunningWI(TrailID, &wiCount, &temp_status))
 973 3     {
 974 3       (void)rbe_user_error(0,
 975 3                            "Internal error: Could not increment
                              running work items for trail, cannot continue.");
 976 3       return -1;
 977 2     }

 979 2     CountOfWorkItemRestoreStarted++;

 981 2     if(0 != GetTQDrivesAcquired(TrailID,
 982 2                                 &DriveAcquiredForTrail,
 983 2                                 &temp_status))
 984 3     {
 985 3       (void)rbe_user_error(0,
 986 3                            "Internal error: Cannot get drives
                                        acquired, cannot continue.");

 988 3       return -1;
 989 2     }

 991 2     if(0 != GetTQDriveConcurrency(TrailID,
 992 2                                   &DriveConcurencyForTrail,
 993 2                                   &temp_status))
 994 3     {
 995 3       (void)rbe_user_error(0,
 996 3                            "Internal error: Cannot get drive
                                          concurrency, cannot continue.");
 997 3       return -1;
 998 2     }

1000 2     *QuitFlag = CancelRestoreTest();

1002 2     if((DriveAcquiredForTrail < DriveConcurencyForTrail) &&
1003 2        ((*CountDrivesInUse) < CountDrivesAvailable) &&
1004 2        (FALSE == *QuitFlag))
1005 3     {
1006 3       continue;
1007 2     }
1008 2     else
1009 3     {
1010 3       break;
1011 2     }
1012 1   }
1013 1   return CountOfWorkItemRestoreStarted;
1014   } /* RunWorkItemRestoresForTrail() */
```

```
1018    /* Stub */
1019    static int DetermineGlobalDriveUse()
1020  1 {
1021  1 /* Limiting to MAXINT === not limiting... Need resource management
1022  1  * to do this properly.
                        NOTE: This should now work like eb_dc_restore does.
1023  1  */
1024  1   return MAXINT;
1025    }
```



```
1028    static int
1029    SendRunningWorkItemsQuit()
1030  1 {
1031  1   int *APlist;
1032  1   int count;
1033  1   int status;
1034  1   int index;

1036  1   if(0 != getPIDList(&count, &APlist, &status))
1037  2   {
1038  2     (void)rbe_user_error(0,
1039  2                          "Internal error: Cannot get auxproc pid list,
                                                                  cannot continue.");
1040  2     return -1;
1041  1   }

1043  1   for(index = 0; index < count; index++)
1044  2   {
1045  2     QuitWorkItemRestore(APlist[index]);
1046  1   }
1047  1   return 0;
1048    }
```

```
1050    /*
1051     * Stub this out for now.
1052     */
1053    static int
1054    InterpretWorkItemRestoreResults(wi_restore_results *results)
1055  1 {
1056  1   return 0;
1057    }
```



```
1059    static void
1060    DebugLogFds(char *error_msg,
1061                fd_set *fds)
1062  1 {
1063  1   int index, fd_count = 0;
1064  1   char buffer[4096];
1065  1   char *bufptr = (char*)buffer;


1068  1   for(index=0;
1069  1       index < 1024;
1070  1       index++)
1071  2   {
1072  2     if( FD_ISSET(index, fds))
1073  3     {
1074  3       int size = 0;
1075  3       size = sprintf(bufptr,"%d,",index);
1076  3       bufptr += size;
1077  3       fd_count++;
1078  2     }
1079  1   }
1080  1   rbe_log_stats(0,"%s fd_count:: %d :: {%s}\n",
1081  1                 error_msg, fd_count, buffer);
1082    }
```

```
1085    static int
1086    test_fd(int fd)
1087  1 {
1088  1   fd_set read_fd;
1089  1   int ret_select;
1090  1   struct timeval timeout = {0, 0};

1092  1   FD_ZERO(&read_fd);

1094  1   FD_SET(fd, &read_fd);

1096  1   do
1097  2   {
1098  2     ret_select = select(fd + 1, &read_fd, NULL, NULL,  &timeout);

1100  1   } while((-1 == ret_select) && (EINTR == errno));

1102  1   return ret_select;

1104    }
```



```
1106    /*
1107     * test_fd_hup()
1108     *
1109     * Description: Test the supplied file descriptor to see if
1110     *   it has had the hang up condition.
1111     *
1112     * Args:
1113     *  Input fd -- the file descriptor to check for the hang up condition.
1114     *
1115     * Returns:
1116     *  1 for HUP event received on fd.
1117     *  0 No HUP event received on fd.
1118     *  -1 errno set.
1119     *
1120     */
1121    static int
1122    test_fd_hup(int fd)
1123  1 {
1124  1   struct pollfd fds;
1125  1   int ret_poll;

1127  1   if(fd < 0)
1128  2   {
1129  2     errno = EINVAL;
1130  2     return -1;
1131  1   }

1133  1   fds.fd = fd;
1134  1   fds.events = POLLIN;
1135  1   fds.revents = 0;  /* initialize */

1137  1   do
1138  2   {
1139  2     ret_poll =  poll(&fds, 1, 0);

1141  1   } while((-1 == ret_poll) && (EINVAL == errno));

1143  1   if(-1 == ret_poll)
1144  2   {
1145  2     return -1;
1146  1   }

1148  1   if(POLLHUP & fds.revents)
1149  2   {
1150  2     return 1;
1151  1   }
1152  1   else
1153  2   {
1154  2     return 0;
1155  1   }

1157    } /* end test_fd_hup() */
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMRESchedApi.cc
 *
 *
 * Mission Statement:  file that contains an API to manage the order
                                                                  in which
 *                     restores occur
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 *                     A few calls to manage the order in which work
                                                                  items
 *                     are run.
                            Currently things are ordered by where they
 *                     appear in the Submit list but this will need to
 *                     change in the future.
 */

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMRESchedApi.cc,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>

#include <restore/REprogmsg.h>
#include <restore/dispatch_daemon.h>

#include <restore/EDMRESubmitApi.h>
#include <EDMREHandleMgrApi.h>
#include <EDMREScheduledWI.h>
#include <EDMRETrailList.h>
#include <EDMRESchedApi.h>

static unsigned int  numberOfQueues = 0;
static RWBinaryTree trailLists;
static pthread_mutex_t    G_scheduleMtx = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    char       templatename[TEMPLNAME_SIZE];
    boolean_ty alternate;
    int        trailnum;
} findArg;

/*************************************************************************
                                                                ********
```



```
**
** Routine:  LockScheduleMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the mutex for the handle tree object
**
*************************************************************************
                                                                ********
*/

static void
LockScheduleMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_scheduleMtx, NULL);
    }

    pthread_mutex_lock(&G_scheduleMtx);
}
```

```
92  /*****************************************************************************
                                                                        ********
93  **
94  ** Routine:  UnlockScheduleMutex
95  **
96  ** Inputs:   None
97  **
98  ** Outputs:  None
99  **
100 ** Return Codes:
101 **           None
102 **
103 ** Purpose:  Unlock the mutex for the handle tree object
104 **
105 *****************************************************************************
                                                                        ********
106 */

108 static void
109 UnlockScheduleMutex()
110 1 {
111 1     pthread_mutex_unlock(&G_scheduleMtx);
112 }
```



```
114 /*****************************************************************************
                                                                        ********
115 **
116 ** Routine:  LookupSchedWI
117 **
118 ** Inputs:   int ID - trail object ID associated with element
119 **           int jobID - wi job ID associated with element
120 **
121 ** Outputs:  int *status - status of the function if an error occured
122 **           EDMRESchedWI **wi - place to put the pointer to the
                                                                        element
123 **
124 ** Return Codes:
125 **           int - 0 for success or non-zero for failure
126 **
127 ** Purpose:  Finds a scheduled work item based on the trail ID and the
128 **           work item ID.
129 **
130 *****************************************************************************
                                                                        ********
131 */

133 static int
134 LookupSchedWI(int ID, int jobID, EDMREScheduledWI **wi,
135                       int *status)
136 1 {
137 1     EDMRETrailList *trl;
138 1     EDMRETrailList *ret;

140 1     if (status == NULL)
141 2     {
142 2         return -1;
143 1     }

145 1     if (wi == NULL)
146 2     {
147 2         *status = SCHED_BAD_PARAM;
148 2         return -1;
149 1     }

151 1     trl = new EDMRETrailList();

153 1     if (trl == NULL)
154 2     {
155 2         *status = SCHED_NO_MEMORY;
156 2         return -1;
157 1     }

159 1     trl -> setTrailQID(ID);

161 1     LockScheduleMutex();

163 1     ret = (EDMRETrailList *) trailLists.find(trl);

165 1     delete trl;

167 1     if (ret == NULL)
168 2     {
169 2         *status = SCHED_TRAIL_LOOKUP_FAILED;
170 2         UnlockScheduleMutex();
171 2         return -1;
172 1     }

174 1     *wi = ret -> getScheduledWI(jobID);
```

```
176 1       UnlockScheduleMutex();

178 1       if (*wi == NULL)
179 2       {
180 2           *status = SCHED_JOB_LOOKUP_FAILED;
181 2           return -1;
182 1       }

184 1       return 0;
185     }
```



```
187     /*************************************************************************
                                                                        ********
188     **
189     ** Routine:  LookupTrailObject
190     **
191     ** Inputs:   int ID - trail object ID
192     **
193     ** Outputs:  int *status - status of the function if an error occured
194     **
195     ** Return Codes:
196     **           int - 0 for success or non-zero for failure
197     **
198     ** Purpose:  Finds a trail object based on the trail ID.
199     **
200     *************************************************************************
                                                                        ********
201     */

203     static int
204     LookupTrailObject(int ID, EDMRETrailList **trl, int *status)
205 1   {
206 1       EDMRETrailList *tmptrl;
207 1       EDMRETrailList *ret;

209 1       if (status == NULL)
210 2       {
211 2           return -1;
212 1       }

214 1       if (trl == NULL)
215 2       {
216 2           *status = SCHED_BAD_PARAM;
217 2           return -1;
218 1       }

220 1       tmptrl = new EDMRETrailList();

222 1       if (tmptrl == NULL)
223 2       {
224 2           *status = SCHED_NO_MEMORY;
225 2           return -1;
226 1       }

228 1       tmptrl -> setTrailQID(ID);

230 1       LockScheduleMutex();

232 1       ret = (EDMRETrailList *) trailLists.find(tmptrl);

234 1       UnlockScheduleMutex();

236 1       delete tmptrl;

238 1       if (ret == NULL)
239 2       {
240 2           *status = SCHED_TRAIL_LOOKUP_FAILED;
241 2           return -1;
242 1       }

244 1       *trl = ret;

246 1       return 0;
247     }
```

```
249  /*****************************************************************************
                                                                          ********
250  **
251  ** Routine:  NewTrailObject
252  **
253  ** Inputs:   NONE
254  **
255  ** Outputs:  int *status - status of the function if an error occured
256  **
257  ** Return Codes:
258  **           int - ID of the new trail Object
259  **
260  ** Purpose:  Creates a new trail object and inserts it in the trail
                                                                      list.
261  **
262  *****************************************************************************
                                                                          ********
263  */

265  int
266  NewTrailObject(int *status)
267 1 {
268 1     EDMRETrailList *tl;
269 1     EDMRETrailList *ret;

271 1     if (status == NULL)
272 2     {
273 2         return 0;
274 1     }

276 1     tl = new EDMRETrailList();

278 1     if (tl == NULL)
279 2     {
280 2         *status = SCHED_BAD_PARAM;
281 2         return 0;
282 1     }

284 1     tl -> setTrailQID(++numberOfQueues);

286 1     LockScheduleMutex();

288 1     ret = (EDMRETrailList *) trailLists.insert(tl);

290 1     UnlockScheduleMutex();

292 1     if (ret == NULL)
293 2     {
294 2         *status = SCHED_TRAIL_INSERT_FAILED;
295 2         delete tl;
296 2         return 0;
297 1     }

299 1     return numberOfQueues;
300  }
```



```
302  /*****************************************************************************
                                                                          ********
303  **
304  ** Routine:  NewSchedWI
305  **
306  ** Inputs:   int ID - trail ID associated with new element
307  **           int submitID - submit ID associated with new element
308  **           int elementID - submit element ID associated with new
                                                                        element
309  **
310  ** Outputs:  int *status - status of the function if an error occured
311  **
312  ** Return Codes:
313  **           int - ID of the new sched WI element
314  **
315  ** Purpose:  Creates a new scheduled work item element and inserts it
                                                                          in the
316  **           trail Q tree.
317  **
318  *****************************************************************************
                                                                          ********
319  */

321  int
322  NewSchedWI(int ID, int submitID, int elementID, int *status)
323 1 {
324 1     EDMRETrailList *trl;
325 1     EDMRETrailList *ret;
326 1     EDMREScheduledWI *wi;
327 1     int    winum = 0;

329 1     if (status == NULL)
330 2     {
331 2         return 0;
332 1     }

334 1     trl = new EDMRETrailList();

336 1     if (trl == NULL)
337 2     {
338 2         *status = SCHED_BAD_PARAM;
339 2         return 0;
340 1     }

342 1     trl -> setTrailQID(ID);

344 1     LockScheduleMutex();

346 1     ret = (EDMRETrailList *) trailLists.find(trl);

348 1     delete trl;

350 1     if (ret == NULL)
351 2     {
352 2         *status = SCHED_TRAIL_LOOKUP_FAILED;
353 2         UnlockScheduleMutex();
354 2         return 0;
355 1     }

357 1     winum = ret -> newScheduledWI();

359 1     if (winum <= 0)
360 2     {
361 2         *status = SCHED_NEW_JOB;
362 2         UnlockScheduleMutex();
```

```
363  2          return 0;
364  1      }

366  1      wi = ret -> getScheduledWI(winum);

368  1      UnlockScheduleMutex();

370  1      if (wi == NULL)
371  2      {
372  2          *status = SCHED_JOB_LOOKUP_FAILED;
373  2          return 0;
374  1      }

376  1      wi -> setSubmitObjectID(submitID);
377  1      wi -> setSubmitElementID(elementID);

379  1      return winum;
380    }
```



```
382  /*****************************************************************************
                                                                          ********
383  **
384  ** Routine:  findTrail
385  **
386  ** Inputs:   TrailList * - TrailList to check against
387  **
388  ** Outputs:  findArg * - structure containing SubmitElement and status
389  **
390  ** Return Codes:
391  **           None
392  **
393  ** Purpose:  Sets the status to non-zero if a match is found.
394  **
395  *****************************************************************************
                                                                          ********
396  */

398  static void
399  findTrail(IN RWCollectable* c, IN void *f)
400  1 {
401  1      EDMRETrailList *trl = (EDMRETrailList *) c;
402  1      findArg        *fa = (findArg *) f;
403  1      char           trltempl[TEMPLNAME_SIZE];

405  1      trl -> getTemplateName(trltempl, sizeof(trltempl));

407  1      if (strcmp(fa->templatename, trltempl) == 0 &&
408  1          fa -> alternate == trl -> isAlternateTrailset())
409  2      {
410  2          fa -> trailnum = trl -> getTrailQID();
411  1      }
412    }
```

```
414  /*****************************************************************
                                                                ********
415  **
416  ** Routine:  GenerateTrailQueues
417  **
418  ** Inputs:   int ID - submit ID
419  **
420  ** Outputs:  int *status - status of the function if an error occured
421  **           int *trailcount - place to put the trail queue count
422  **
423  ** Return Codes:
424  **           int - 0 if success and non-zero otherwise
425  **
426  ** Purpose:  Generate the schedule ordered by trail.
427  **
428  *****************************************************************
                                                                ********
429  */

431  int
432  GenerateTrailQueues(int ID, int *trailcount, int *status)
433 1 {
434 1     EDMRESubmitObj *so;
435 1     EDMRESubmitElement *se;
436 1     EDMRETrailList *trl;
437 1     int ret;
438 1     int i;
439 1     int numElements;
440 1     findArg      fa;

442 1     if (status == NULL)
443 2     {
444 2         return -1;
445 1     }

447 1     if (trailcount == NULL)
448 2     {
449 2         *status = SCHED_BAD_PARAM;
450 2         return -1;
451 1     }

453 1     ret = LookupSubmitObject(ID, &so, status);

455 1     if (ret != 0)
456 2     {
457 2         return ret;
458 1     }

460 1     numElements = so -> getWICount();

462 1     for (i = 1; i < numElements + 1; i++)
463 2     {
464 2         se = so -> getSubmitElement(i);

466 2         if (se == NULL)
467 3         {
468 3             *status = SCHED_SE_LOOKUP_FAILURE;
469 3             break;
470 2         }

472 2         fa.alternate = se -> IsAlternateTrailset();
473 2         se -> getTemplate(fa.templatename, TEMPLNAME_SIZE);
474 2         fa.trailnum = 0;

476 2         LockScheduleMutex();
```



```
478 2         trailLists.apply(findTrail, &fa);

480 2         UnlockScheduleMutex();

482 2         if (fa.trailnum == 0)
483 3         {
484 3             char  setempl[TEMPLNAME_SIZE];

486 3             fa.trailnum = NewTrailObject(status);

488 3             if (fa.trailnum == 0)
489 4             {
490 4                 break;
491 3             }

493 3             ret = LookupTrailObject(fa.trailnum, &trl, status);

495 3             if (ret != 0)
496 4             {
497 4                 break;
498 3             }

500 3             trl -> setSubmitID(ID);
501 3             trl -> setAlternateTrailset(se -> IsAlternateTrailset());

503 3             se -> getTemplate(setempl, sizeof(setempl));
504 3             trl -> setTemplateName(setempl);
505 2         }

507 2         ret = NewSchedWI(fa.trailnum, ID, i, status);

509 2         if (ret <= 0)
510 3         {
511 3             break;
512 2         }
513 1     }

515 1     *trailcount = numberOfQueues;

517 1     return 0;
518  }
```

```
520  /*****************************************************************************
                                                                          ********
521  **
522  ** Routine:  PurgeTrailQueue
523  **
524  ** Inputs:   None
525  **
526  ** Outputs:  None
527  **
528  ** Return Codes:
529  **           None
530  **
531  ** Purpose:  Purge all lists.
532  **
533  *****************************************************************************
                                                                          ********
534  */

536  void
537  PurgeTrailQueue()
538 1 {
539 1     LockScheduleMutex();

541 1     trailLists.clearAndDestroy();

543 1     UnlockScheduleMutex();

545 1     numberOfQueues = 0;
546  }
```



```
548  /*****************************************************************************
                                                                          ********
549  **
550  ** Routine:  ActivateTrailQueue
551  **
552  ** Inputs:   int ID - trail ID
553  **           int drivecount - number of drives to use
554  **
555  ** Outputs:  int *status - status of the function if an error occured
556  **
557  ** Return Codes:
558  **           int - 0 if success and non-zero otherwise
559  **
560  ** Purpose:  Active the given trail for restore.
561  **
562  *****************************************************************************
                                                                          ********
563  */

565  int
566  ActivateTrailQueue(int ID, int drivecount, int *status)
567 1 {
568 1     EDMRETrailList *trl;
569 1     int  ret = 0;

571 1     if (status == NULL)
572 2     {
573 2         return -1;
574 1     }

576 1     ret = LookupTrailObject(ID, &trl, status);

578 1     if (ret != 0)
579 2     {
580 2         return ret;
581 1     }

583 1     if (trl == NULL)
584 2     {
585 2         return -1;
586 1     }

588 1     trl -> setTrailActive(TRUE);
589 1     trl -> setMaxDrives(drivecount);

591 1     return 0;
592  }
```

```
/*****************************************************************************
                                                                        ********
**
** Routine:  DeactivateTrailQueue
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Deactivate the given trail for restore.
**
*****************************************************************************
                                                                        ********
*/

int
DeactivateTrailQueue(int ID, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setTrailActive(FALSE);

    return 0;
}
```



```
/*****************************************************************************
                                                                        ********
**
** Routine:  SetTQDriveConcurrency
**
** Inputs:   int ID - trail ID
**           int drivecount - number of drives to use
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Set the drive concurrency for the given trail.
**
*****************************************************************************
                                                                        ********
*/

int
SetTQDriveConcurrency(int ID, int drivecount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setMaxDrives(drivecount);

    return 0;
}
```

```
683  /*****************************************************************************
                                                                          ********
684  **
685  ** Routine:  GetTQDriveConcurrency
686  **
687  ** Inputs:   int ID - trail ID
688  **
689  ** Outputs:  int *status - status of the function if an error occured
690  **           int *drivecount - place for the number of drives to use
691  **
692  ** Return Codes:
693  **           int - 0 if success and non-zero otherwise
694  **
695  ** Purpose:  Retrieve the number of drives to use for the given trail
696  **           for restore.
697  **
698  *****************************************************************************
                                                                          ********
699  */

701  int
702  GetTQDriveConcurrency(int ID, int *drivecount, int *status)
703 1 {
704 1     EDMRETrailList *trl;
705 1     int  ret = 0;

707 1     if (status == NULL)
708 2     {
709 2         return -1;
710 1     }

712 1     if (drivecount == NULL)
713 2     {
714 2         *status = SCHED_BAD_PARAM;
715 2         return -1;
716 1     }

718 1     ret = LookupTrailObject(ID, &trl, status);

720 1     if (ret != 0)
721 2     {
722 2         return ret;
723 1     }

725 1     if (trl == NULL)
726 2     {
727 2         return -1;
728 1     }

730 1     *drivecount = trl -> getMaxDrives();

732 1     return 0;
733  }
```



```
735  /*****************************************************************************
                                                                          ********
736  **
737  ** Routine:  SetTQDrivesAcquired
738  **
739  ** Inputs:   int ID - trail ID
740  **           int drivecount - the number of drives in use
741  **
742  ** Outputs:  int *status - status of the function if an error occured
743  **
744  ** Return Codes:
745  **           int - 0 if success and non-zero otherwise
746  **
747  ** Purpose:  Sets the number of drives in use for the given trail
748  **           for restore.
749  **
750  *****************************************************************************
                                                                          ********
751  */

753  int
754  SetTQDrivesAcquired(int ID, int drivecount, int *status)
755 1 {
756 1     EDMRETrailList *trl;
757 1     int  ret = 0;

759 1     if (status == NULL)
760 2     {
761 2         return -1;
762 1     }

764 1     ret = LookupTrailObject(ID, &trl, status);

766 1     if (ret != 0)
767 2     {
768 2         return ret;
769 1     }

771 1     if (trl == NULL)
772 2     {
773 2         return -1;
774 1     }

776 1     trl -> setDrivesInUse(drivecount);

778 1     return 0;
779  }
```

```
781  /*****************************************************************************
                                                                          ********
782  **
783  ** Routine:  GetTQDrivesAcquired
784  **
785  ** Inputs:   int ID - trail ID
786  **
787  ** Outputs:  int *status - status of the function if an error occured
788  **           int *drivecount - a place to put the number of drives in
                                                                             use
789  **
790  ** Return Codes:
791  **           int - 0 if success and non-zero otherwise
792  **
793  ** Purpose:  Gets the number of drives in use for the given trail
794  **           for restore.
795  **
796  *****************************************************************************
                                                                          ********
797  */

799  int
800  GetTQDrivesAcquired(int ID, int *drivecount, int *status)
801 1 {
802 1     EDMRETrailList *trl;
803 1     int  ret = 0;

805 1     if (status == NULL)
806 2     {
807 2         return -1;
808 1     }

810 1     if (drivecount == NULL)
811 2     {
812 2         *status = SCHED_BAD_PARAM;
813 2         return -1;
814 1     }

816 1     ret = LookupTrailObject(ID, &trl, status);

818 1     if (ret != 0)
819 2     {
820 2         return ret;
821 1     }

823 1     if (trl == NULL)
824 2     {
825 2         return -1;
826 1     }

828 1     *drivecount = trl -> getDrivesInUse();

830 1     return 0;
831  }
```



```
833  /*****************************************************************************
                                                                          ********
834  **
835  ** Routine:  IncrementRunningWI
836  **
837  ** Inputs:   int ID - trail ID
838  **
839  ** Outputs:  int *wiCount - number of work items running after
                                                                        increment.
840  **           int *status - status of the function if an error occured
841  **
842  ** Return Codes:
843  **           int - 0 if success and non-zero otherwise
844  **
845  ** Purpose:  Increment the running work items for the given trail.
846  **
847  *****************************************************************************
                                                                          ********
848  */

850  int
851  IncrementRunningWI(int ID, int *wiCount, int *status)
852 1 {
853 1     EDMRETrailList *trl;
854 1     int  ret = 0;

856 1     if (status == NULL)
857 2     {
858 2         return -1;
859 1     }

861 1     if (wiCount == NULL)
862 2     {
863 2         *status = SCHED_BAD_PARAM;
864 2         return -1;
865 1     }

867 1     ret = LookupTrailObject(ID, &trl, status);

869 1     if (ret != 0)
870 2     {
871 2         return ret;
872 1     }

874 1     if (trl == NULL)
875 2     {
876 2         return -1;
877 1     }

879 1     *wiCount = trl -> IncrementRunningWIs();

881 1     return 0;
882  }
```

```
884  /*****************************************************************************
                                                                          ********
885  **
886  ** Routine:  DecrementRunningWI
887  **
888  ** Inputs:   int ID - trail ID
889  **
890  ** Outputs:  int *wiCount - number of work items running after
                                                                        decrement.
891  **           int *status - status of the function if an error occured
892  **
893  ** Return Codes:
894  **           int - 0 if success and non-zero otherwise
895  **
896  ** Purpose:  Decrement the running work items for the given trail.
897  **
898  *****************************************************************************
                                                                          ********
899  */

901  int
902  DecrementRunningWI(int ID, int *wiCount, int *status)
903 1 {
904 1     EDMRETrailList *trl;
905 1     int  ret = 0;

907 1     if (status == NULL)
908 2     {
909 2         return -1;
910 1     }

912 1     if (wiCount == NULL)
913 2     {
914 2         *status = SCHED_BAD_PARAM;
915 2         return -1;
916 1     }

918 1     ret = LookupTrailObject(ID, &trl, status);

920 1     if (ret != 0)
921 2     {
922 2         return ret;
923 1     }

925 1     if (trl == NULL)
926 2     {
927 2         return -1;
928 1     }

930 1     *wiCount = trl -> DecrementRunningWIs();

932 1     return 0;
933  }
```



```
934  /*****************************************************************************
                                                                          ********
935  **
936  ** Routine:  SetRunningWI
937  **
938  ** Inputs:   int ID - trail ID
939  **           int wiCount - number of work items to set.
940  **
941  ** Outputs:  int *status - status of the function if an error occured
942  **
943  ** Return Codes:
944  **           int - 0 if success and non-zero otherwise
945  **
946  ** Purpose:  Set the running work items for the given trail.
947  **
948  *****************************************************************************
                                                                          ********
949  */

951  int
952  SetRunningWI(int ID, int wiCount, int *status)
953 1 {
954 1     EDMRETrailList *trl;
955 1     int  ret = 0;

957 1     if (status == NULL)
958 2     {
959 2         return -1;
960 1     }

962 1     ret = LookupTrailObject(ID, &trl, status);

964 1     if (ret != 0)
965 2     {
966 2         return ret;
967 1     }

969 1     if (trl == NULL)
970 2     {
971 2         return -1;
972 1     }

974 1     trl -> setRunningWIs(wiCount);

976 1     return 0;
977  }
```

```
978  /*****************************************************************************
                                                                          ********
979  **
980  ** Routine:  GetRunningWI
981  **
982  ** Inputs:   int ID - trail ID
983  **
984  ** Outputs:  int *wiCount - number of work items running.
985  **           int *status - status of the function if an error occured
986  **
987  ** Return Codes:
988  **           int - 0 if success and non-zero otherwise
989  **
990  ** Purpose:  Get the running work items for the given trail.
991  **
992  *****************************************************************************
                                                                          ********
993  */
994  int
995  GetRunningWI(int ID, int *wiCount, int *status)
996 1 {
997 1     EDMRETrailList *trl;
998 1     int  ret = 0;

1000 1    if (status == NULL)
1001 2    {
1002 2        return -1;
1003 1    }

1005 1    if (wiCount == NULL)
1006 2    {
1007 2        *status = SCHED_BAD_PARAM;
1008 2        return -1;
1009 1    }

1011 1    ret = LookupTrailObject(ID, &trl, status);

1013 1    if (ret != 0)
1014 2    {
1015 2        return ret;
1016 1    }

1018 1    if (trl == NULL)
1019 2    {
1020 2        return -1;
1021 1    }

1023 1    *wiCount = trl -> getRunningWIs();

1025 1    return 0;
1026 }
```



```
1029 /*****************************************************************************
                                                                          ********
1030 **
1031 ** Routine:  PopWIFromTrailQueue
1032 **
1033 ** Inputs:   int ID - trail ID
1034 **
1035 ** Outputs:  int *status - status of the function if an error occured
1036 **           int *submitID - a place to put the submitID
1037 **           int *elementID - a place to put the element ID
1038 **
1039 ** Return Codes:
1040 **           int - 0 if success and non-zero otherwise
1041 **
1042 ** Purpose:  Gets the submit ID and element ID of the next work item
                                                                        to run.
1043 **
1044 *****************************************************************************
                                                                          ********
1045 */

1047 int
1048 PopWIFromTrailQueue(
                    int ID, int *submitID, int *elementID, int *status)
1049 1 {
1050 1     EDMRETrailList *trl;
1051 1     EDMRESchedWI *sw;
1052 1     int  ret = 0;

1054 1     if (status == NULL)
1055 2     {
1056 2         return -1;
1057 1     }

1059 1     if (submitID == NULL || elementID == NULL)
1060 2     {
1061 2         *status = SCHED_BAD_PARAM;
1062 2         return -1;
1063 1     }

1065 1     ret = LookupTrailObject(ID, &trl, status);

1067 1     if (ret != 0)
1068 2     {
1069 2         return ret;
1070 1     }

1072 1     if (trl == NULL)
1073 2     {
1074 2         return -1;
1075 1     }

1077 1     if (trl -> isTrailActive() == FALSE)
1078 2     {
1079 2         *status = SCHED_TRAIL_NOT_ACTIVE;
1080 2         return -1;
1081 1     }

1083 1     sw = trl -> popScheduledWI();

1085 1     if (sw == NULL)
1086 2     {
1087 2         *status = SCHED_NO_MORE_JOBS;
1088 2         return -1;
```

```
1089  1      }

1091  1      *submitID = sw -> getSubmitObjectID();
1092  1      *elementID = sw -> getSubmitElementID();

1094  1      delete sw;

1096  1      return 0;
1097    }
```



```
1099    /*****************************************************************************
                                                                              ********
1100    **
1101    ** Routine:  AddWIToTrailQueue
1102    **
1103    ** Inputs:   int ID - trail ID
1104    **           int submitID - the submitID of the work item
1105    **           int elementID - the element ID of the work item
1106    **
1107    ** Outputs:  int *status - status of the function if an error occured
1108    **
1109    ** Return Codes:
1110    **           int - 0 if success and non-zero otherwise
1111    **
1112    ** Purpose:  Add the work item described by the submit ID and element
                                                                                 ID to
1113    **           the specified trail queue.
1114    **
1115    *****************************************************************************
                                                                              ********
1116    */

1118    int
1119    AddWIToTrailQueue(int ID, int submitID, int elementID, int *status)
1120  1 {
1121  1     int ret = 0;

1123  1     if (status == NULL)
1124  2     {
1125  2         return -1;
1126  1     }

1128  1     ret = NewSchedWI(ID, submitID, elementID, status);

1130  1     if (ret <= 0)
1131  2     {
1132  2         return -1;
1133  1     }

1135  1     return 0;
1136    }
```

```
1138  /*************************************************************************
                                                                        ********
1139  **
1140  ** Routine:  FindTrailQueueOfWI
1141  **
1142  ** Inputs:   int handle - handle to identify work item
1143  **
1144  ** Outputs:  int *status - status of the function if an error occured
1145  **           int *ID - a place to put the trail ID
1146  **
1147  ** Return Codes:
1148  **           int - 0 if success and non-zero otherwise
1149  **
1150  ** Purpose:  Gets the trail ID of the work item.
1151  **
1152  *************************************************************************
                                                                        ********
1153  */

1155  int
1156  FindTrailQueueOfWI(int handle, int *ID, int *status)
1157 1 {
1158 1     EDMRESubmitElement *se;
1159 1     int                 soID,
1160 1                         seID;
1161 1     int                 ret;
1162 1     findArg             fa;

1164 1     if (status == NULL)
1165 2     {
1166 2         return -1;
1167 1     }

1169 1     if (ID == NULL)
1170 2     {
1171 2         *status = SCHED_BAD_PARAM;
1172 2         return -1;
1173 1     }

1175 1     ret = getSubmitIDs(handle, &soID, &seID, status);

1177 1     if (ret != 0)
1178 1         return ret;

1180 1     ret = LookupSubmitElement(soID, seID, &se, status);

1182 1     if (ret != 0)
1183 1         return ret;

1185 1     fa.alternate = se -> IsAlternateTrailset();
1186 1     se -> getTemplate(fa.templatename, TEMPLNAME_SIZE);
1187 1     fa.trailnum = 0;

1189 1     LockScheduleMutex();

1191 1     trailLists.apply(findTrail, &fa);

1193 1     UnlockScheduleMutex();

1195 1     if (fa.trailnum == 0)
1196 1         return -1;

1198 1     *ID = fa.trailnum;

1200 1     return 0;
```



```
1201  }
```